Parallel Backtracking with Answer Memoing for Independent And-Parallelism∗

Pablo Chico de Guzmán,[1] Amadeo Casas,[2] Manuel Carro,[1,3] and Manuel V. Hermenegildo[1,3]

[1] *School of Computer Science, Univ. Politécnica de Madrid, Spain.*
(*e-mail:* pchico@clip.dia.fi.upm.es, {mcarro,herme}@fi.upm.es)
[2] *Samsung Research, USA.*
(*e-mail:* amadeo.c@samsung.com)
[3] *IMDEA Software Institute, Spain.*
(*e-mail:* {manuel.carro,manuel.hermenegildo}@imdea.org)

Abstract

Goal-level Independent and-parallelism (IAP) is exploited by scheduling for simultaneous execution two or more goals which will not interfere with each other at run time. This can be done safely even if such goals can produce multiple answers. The most successful IAP implementations to date have used recomputation of answers and sequentially ordered backtracking. While in principle simplifying the implementation, recomputation can be very inefficient if the granularity of the parallel goals is large enough and they produce several answers, while sequentially ordered backtracking limits parallelism. And, despite the expected simplification, the implementation of the classic schemes has proved to involve complex engineering, with the consequent difficulty for system maintenance and extension, while still frequently running into the well-known trapped goal and garbage slot problems. This work presents an alternative parallel backtracking model for IAP and its implementation. The model features parallel out-of-order (i.e., non-chronological) backtracking and relies on answer memoization to reuse and combine answers. We show that this approach can bring significant performance advantages. Also, it can bring some simplification to the important engineering task involved in implementing the backtracking mechanism of previous approaches.

KEYWORDS: Parallelism, Logic Programming, Memoization, Backtracking, Performance.

1 Introduction

Widely available multicore processors have brought renewed interest in languages and tools to efficiently and transparently exploit parallel execution — i.e., tools to take care of the difficult (Karp and Babb 1988) task of automatically uncovering parallelism in sequential algorithms and in languages to succinctly express this parallelism. These languages can be used to both write directly parallel applications and as targets for parallelizing compilers.

Declarative languages (and among them, logic programming languages) have traditionally been considered attractive for both expressing and exploiting parallelism due to their clean and simple semantics and their expressive power. A large amount of work has been done in the area of parallel execution of logic programs (Gupta et al. 2001), where two main sources of parallelism have been exploited: parallelism between goals of a resolvent (And-Parallelism) and parallelism between the branches of the execution (Or-Parallelism). Systems efficiently exploiting Or-Parallelism include Aurora (Lusk et al. 1988) and MUSE (Ali and Karlsson 1990),

∗ Work partially funded by EU projects IST-215483 *S-Cube* and FET IST-231620 *HATS*, MICINN projects TIN-2008-05624 *DOVES*, and CAM project S2009TIC-1465 *PROMETIDOS*. Pablo Chico is also funded by an MICINN FPU scholarship.

while among those exploiting And-Parallelism, &-Prolog (Hermenegildo and Greene 1991) and DDAS (Shen 1996) are among the best known ones. In particular, &-Prolog exploits *Independent And-Parallelism*, where goals to be executed in parallel do not compete for bindings to the same variables at run time and are launched following a nested fork-join structure. Other systems such as (&)ACE (Pontelli et al. 1995), AKL (Janson 1994), Andorra-I (Santos-Costa 1993) and the Extended Andorra Model (EAM) (Santos Costa, V. et al. 1991; Lopes et al. 2011) have approached a combination of both or- and and-parallelism. In this paper, we will focus on independent and-parallelism.

While many IAP implementations obtained admirable performance results and achieved efficient memory management, implementing synchronization and working around problems such as *trapped goals* (Section 5) and *garbage slots* in the execution stacks required complex engineering: extensions to the WAM instruction set, new data structures, special stack frames in the stack sets, and others (Hermenegildo 1986). Due to this complexity, recent approaches have focused instead on simplicity, moving core components of the implementation to the source level. In (Casas et al. 2008), a high-level implementation of goal-level IAP was proposed that showed reasonable speedups despite the overhead added by the high level of the implementation. Other recent proposals (Moura et al. 2008), with a different focus than the traditional approaches to parallelism in LP, concentrate on providing machinery to take advantage of underlying thread-based OS building blocks.

A critical area in the context of IAP that has also received much attention is the implementation of backtracking. Since in IAP by definition goals do not affect each other, an obvious approach is to generate all the solutions for these goals in parallel independently, and then combine them (Conery 1987). However, this approach has several drawbacks. First, copying solutions, at least naively, can imply very significant overhead. In addition, this approach can perform an unbounded amount of unnecessary work if, e.g., only some of the solutions are actually needed, and it can even be non-terminating if one of the goals does not fail finitely. For these reasons the operational semantics typically implemented in IAP systems performs an ordered, right-to-left backtracking. For example, if execution backtracks into a parallel conjunction such as `a & b & c`, the rightmost goal (`c`) backtracks first. If it fails, then `b` is backtracked over while `c` is recomputed and so on, until a new solution is found or until the parallel conjunction fails. The advantage of this approach is that it saves memory (since no solutions need to be copied) and keeps close to the sequential semantics. However, it also implies that many computations are redone and a large amount of backtracking work can be essentially sequential.

Herein we propose an improved solution to backtracking in IAP aimed at reducing recomputation and increasing parallelism while preserving efficiency. It combines *memoization of answers to parallel goals* (to avoid recomputation), *out-of-order backtracking* (to exploit parallelism on backtracking), and *incremental computation of answers*, to reduce memory consumption and avoid termination problems. The fact that in this approach the right-to-left rule may not be followed during parallel backtracking means that answer generation order can be affected (this of course does not affect the declarative semantics) but, as explained later, it greatly simplifies implementation. The EAM also supports out-of-order execution of goals. However, our approach differs from EAM in that the EAM is a more encompassing and complex approach, offering more parallelism at the cost of more complexity (and overhead) while our proposal constitutes a simpler and more approachable solution to implement.

In the following we present our proposal and an IAP implementation of the approach, and we provide experimental data showing that the amount of parallelism exploited increases due to the parallelism in backward execution, while keeping competitive performance for first-answer queries. We also observe super-linear speedups, achievable thanks to memoization of previous answers (which are recomputed in sequential SLD resolution).[1]

2 An Overview of IAP with Parallel Backtracking

In this section we provide a high-level view of the execution algorithm we propose to introduce some concepts which we will explain in more detail in later sections.

The IAP + parallel backtracking model we propose behaves in many respects as classical IAP approaches, but it has as its main difference the use of speculative backward execution (when possible) to generate additional solutions eagerly. This brings a number of additional changes which have to be accommodated. We assume as usual in IAP a number of *agents*, which are normally each attached to their own *stack set*, composed of heap, trail, stack, and goal queue (and often referred in the following simply as a “stack”). Active agents are executing code using their stack set, and they place any new parallel work they find in their goal queue. Idle agents steal parallel work from the goal queues of other agents.[2] We will also assume that stack sets have a new memo area for storing solutions (explained further later, see Figure 2).

Forward execution: as in classical IAP, when a parallel conjunction is first reached, its goals are started in parallel. When a goal in the conjunction fails without returning any solution, the whole conjunction fails. And when all goals have found a solution, execution proceeds. However, and differently to classical IAP, if a solution has been found for some goals, but not for all, the agents which did finish may speculatively perform backward execution for the goals they executed (unless there is a need for agents to execute work which is not speculative, e.g., to generate the first answer to a goal). This in turn brings the need to stash away the generated solutions in order to continue searching for more answers (which are also saved). When all goals find a solution, those which were speculatively executing are suspended (to preserve the property of no-slowdown w.r.t. sequential execution (Hermenegildo and Rossi 1995)), their state is saved to be resumed later, and their first answer is reinstalled.

Backward execution: we only perform backtracking on the goals of a parallel conjunction which are on top of the stacks. If necessary, stack sections are reordered to move trapped goals to the top of the stack. In order not to impose a rigid ordering, we allow backtracking on these goals to proceed in an arbitrary order (i.e., not necessarily corresponding to the lexical right-to-left order). This opens the possibility of performing backtracking in parallel, which brings some additional issues to take care of:

- When some of the goals executing backtracking in parallel find a new answer, backtracking stops by suspending the rest of the goals and saving their state.
- The solution found is saved in the memoing area, in order to avoid recomputation.

[1] For brevity we assume some familiarity with the WAM (Warren 1983; Ait-Kaci 1991) and the RAP-WAM (Hermenegildo and Greene 1991).
[2] For a more in-depth understanding of the memory model and scheduling used in traditional IAP approaches, please refer to (Hermenegildo and Greene 1991; Shen and Hermenegildo 1996; Gupta et al. 2001).

- Every new solution is combined with the previously available solutions. Some of these will be recovered from the memoization memory and others may simply be available if they are the last solution computed by some goal and thus the bindings are active.
- If more solutions are needed, backward execution is performed in parallel again. Goals which were suspended resume where they suspended.

All this brings the necessity of saving and resuming execution states, memoing and recovering answers quickly, combining previously existing solutions with newly found solutions, assigning agents to speculative computations only if there are no non-speculative computations available, and managing computations which change from speculative to non speculative. Note that all parallel backtracking is speculative work, because we might need just one more answer of the rightmost parallel goal, and this is why backward execution is given less priority than forward execution. Note also that at any point in time we only have one active value for each variable. While performing parallel backtracking we can change the bindings which will be used in forward execution, but before continuing with forward execution, all parallel goals have to suspend to reinstall the bindings of the answer being combined.

3 An Execution Example

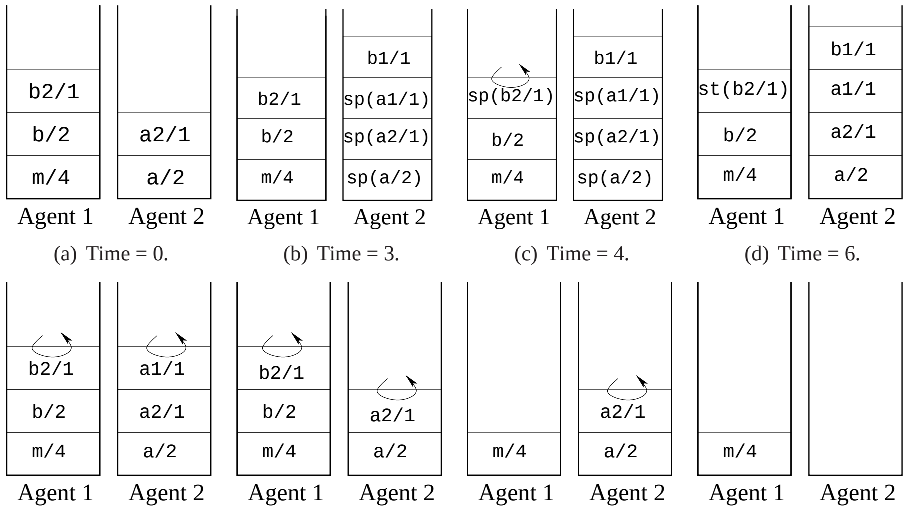
We will illustrate our approach, and specially the interplay of memoization and parallel backtracking in IAP execution with the following program:

```
main(X, Y, Z, T)  :- a(X, Y) & b(Z, T).
a(X, Y)  :- a1(X) & a2(Y).
b(X, Y)  :- b1(X) & b2(Y).
```

We will assume that a1(X), a2(Y), b1(X) and b2(Y) have two answers each, which take 1 and 7 seconds, 2 and 10 seconds, 3 and 13 seconds, and 4 and 25 seconds, respectively. We will also assume that there are no dependencies among the variables in the literals of these clauses, and that the cost of preparing and starting up parallel goals is negligible. Finally, we will assume that there are two agents available to execute these goals at the beginning of the execution of the predicate main/4. Figure 1 summarizes the evolution of the stack of each agent throughout the execution of main/4 (abbreviated as m/4 in the figure).

Once the first agent starts the execution of main/4, a/2 is published for parallel execution and b/2 is executed locally. The second agent steals a/2, publishes a1/1 for parallel execution and executes a2/1 locally, while the first agent marks b1/1 as parallel and executes b2/1. The execution state can be seen in Figure 1(a). When the second agent finds the first answer for a2/1, it marks a2/1 to be executed in a speculative manner. However, since a1/1 and b1/1 are still pending, the second agent will start executing one of them instead. We will assume it starts executing a1/1. Once it finds an answer, a1/1 is marked to be executed speculatively. Since a2/1 is also marked as such, then the entire predicate a/2 can be configured to be executed speculatively. However, the second agent will now execute b1/1 since it is pending and has higher priority than speculative execution (Figure 1(b)).

Figure 1(c) shows the execution state when the first agent finds an answer for b2/1. In this case, since there is no other parallel goal to execute, the first agent starts the execution of b2/1 speculatively, until the second agent finishes the execution of b1/1. When that happens, the first agent suspends the execution of b2/1 and the first answer of main/4 is returned, as shown in Figure 1(d).



(a) Time = 0. (b) Time = 3. (c) Time = 4. (d) Time = 6.

(e) Time = 16. (f) Time = 23. (g) Time = 29. (h) Time = 36.

Fig. 1. Execution of main/4 with memoization of answers and parallel backtracking.

In order to calculate the next answer of main/4, both agents will backtrack over b2/1 and b1/1, respectively. Note that they would not be able to backtrack over other subgoals because they are currently trapped. Once the second agent finds the second answer of b1/1, the first agent suspends the execution of b2/1 and returns the second answer of main/4, combining all the existing answers of its literals.

In order to obtain the next answer of main/4, the first agent continues with the execution of b2/1, and the second agent fails the execution of b1/1 and starts computing the next answer of a1/1, since that goal has now been freed, as shown in Figure 1(e). Whenever the answer of a1/1 is completed, shown in Figure 1(f), the execution of b2/1 is again suspended and a set of new answers of main/4 involving the new answer for a2/1 can be returned, again as a combination of the already computed answers of its subgoals. To obtain the rest of the answers of predicate main/4, the first agent resumes the execution of b2/1 and the second agent starts calculating a new answer of a2/1 (Figure 1(g)). The first agent finds the answer of b2/1, suspends the execution of the second agent, and returns the new answers of main/4. Finally, Figure 1(h) shows how the second agent continues with the execution of a2/1 in order to obtain the rest of the answers of main/4.

Note that in this example memoization of answers avoids having to recompute expensive answers of parallel goals. Also note that all the answers for each parallel literal could have been found separately and then merged, producing a similar total execution time. However, the computational time for the first answer would have been drastically increased.

4 Memoization vs. Recomputation

Classic IAP uses recomputation of answers: if we execute a(X) & b(Y), the first answer of each goal is generated in parallel. On backtracking, b(Y) generates additional answers (one by one, sequentially) until it finitely fails. Then, a new answer for goal a(X) is computed in parallel with the recomputation of the first answer of b(Y). Successive answers are computed by backtracking again on b(Y), and later on a(X).

However, since a(X) and b(Y) are independent, the answers of goal b(Y) will be the same

in each recomputation. Consequently, it makes sense to store its bindings after every answer is generated, and combine them with those from a(X) to avoid the recomputation of b(Y). Memoing answers does not require having the bindings for these answers on the stack; in fact they should be stashed away and reinstalled when necessary. Therefore, when a new answer is computed for a(X) the previously computed and memorized answers for b(Y) are restored and combined.

4.1 Answer Memoization

In comparison with tabling (Tamaki and Sato 1986; Warren 1992; Chen and Warren 1996), which also saves goal answers, our scheme shows a number of differences: we assume that we start off with terminating programs (or that if the original program is non-terminating in sequential Prolog, we do not need to terminate), and therefore we do not need to take care of the cases tabling has to: detecting repeated calls,[3] suspending / resuming consumers, maintaining SCCs, etc. We do not keep stored answers after a parallel call finitely fails: answers for a(X) & b(Y) are kept for only as long as the new bindings for X and Y are reachable. In fact, we can discard *all* stored answers as soon as the parallel conjunction continues after its last answer. Additionally, we restrict the visibility of the stored answers to the parallel conjunction: if we have a(X) & b(Y), a(Z), the calls to a(Z) do not have access to the answers for a(X). While this may lead to underusing the saved bindings, it greatly simplifies the implementation and reduces the associated overhead. Therefore we will not use the memoization machinery commonly found in tabling implementations (Ramakrishnan et al. 1995).

Instead, we save a combination of trail and heap terms which capture all the bindings made by the execution of a goal, for which we need two slight changes: we push a choicepoint before the parallel goal execution, so that all bindings to variables which live before the parallel goal execution will be recorded, and we modify the trail code to always trail variables which are not in the agent's WAM.[4] This ensures that all variable bindings we need to save are recorded on the trail.

Therefore what we need to save are the variables pointed from the trail segment corresponding to the execution of the parallel goal (where the bindings to its free variables are recorded) and the terms pointed to by these variables. These terms are only saved if they live in the heap segment which starts after the execution of the parallel goal, since if they live below that point they existed before the parallel goal was executed and they are unaffected by backtracking. Note that bindings to variables which were created *within* the execution of the parallel goal and which are not reachable from the argument variables do not have to be recorded, as they are not visible outside the scope of the parallel goal execution.[5]

Figure 2 shows an example. G is a parallel goal whose execution unifies: X with a list existing before the execution of G, Y with a list created by G, and Z, which was created by G, with a list also created by G. Consequently, we save those variables appearing in the trail created by G which are older than the execution of G (X and Y), and all the structures

[3] Detecting repeated calls requires traversing the arguments of a goal, which can be arbitrarily more costly than executing the goal itself: for example, consider taking a large list and returning just its first element, as in first([X|_],X).

[4] This introduces a slight overhead which we have measured at around 1%.

[5] Another possible optimization is to share bindings corresponding to common parts of the search tree of a parallel goal: if a new answer is generated by performing backtracking on, for example, the topmost choicepoint and the rest of the bindings generated by the goal are not changed, strictly speaking only these different bindings have to be saved to save the new answer, and not the whole section of trail and heap.

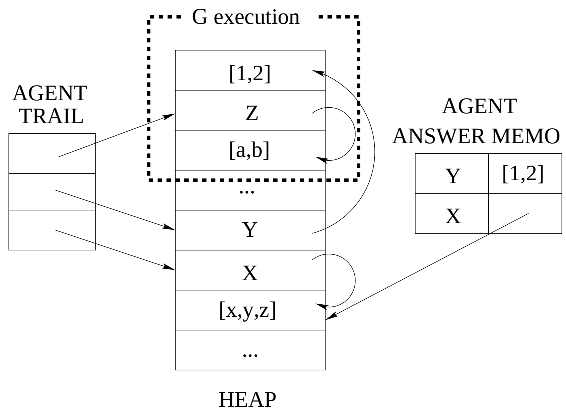


Fig. 2. Snapshot of agent's stacks during answer memoization process.

hanging from them. [x,y,z] is not copied because is not affected by backtracking. The copy operation adjusts pointers of variables in a way that is similar to what is done in tabling implementations (Ramakrishnan et al. 1995). For example, if we save a variable pointing to a subterm of [1,2], this variable would now point to a subterm of the copy of [1,2].

Note that this is at most the same amount of work as that of the execution of the goal, because it consists of stashing away the variables bound by the goal plus the structures *created* by the goal. The information related to the boundaries of the goal and its answers is kept in a centralized per-conjunction data structure, akin to a *parcall frame* (Hermenegildo and Greene 1991). Similar techniques are also used for the local stack.

Reinstalling an answer for a goal boils down to copying back to the heap the terms that were previously saved and using the trail entries to make the variables in the initial call point to the terms they were bound to when the goal had finished. Some of these variables point to the terms just copied onto the heap and some will point to terms which existed previously to the goal execution and which were therefore not saved. In our example, [1,2] is copied onto the heap and unified with Y and X is unified with [x,y,z], which was already living on the heap.

As mentioned before, while memoization certainly has a cost, it can also provide by itself substantial speedups since it avoids recomputations. Since it is performed only on *independent* goals, the number of different solutions to keep does not grow exponentially with the number of goals in a conjunction, but rather only linearly. This is an interesting case of synergy between two different concepts (independence and memoization), which in principle are orthogonal, but which happen to have a very positive mutual interaction.

4.2 Combining Answers

When the last goal pending to generate an answer in a parallel conjunction produces a solution, any sibling goals which were speculatively working towards producing additional solutions have to suspend, reinstall the previously found answers, and combine them to continue with forward execution. A similar behavior is necessary when backtracking is performed over a parallel conjunction and one of the goals which are being reexecuted in parallel finds a new solution. At this moment, the new answer is combined with all the previous answers of the rest of the parallel goals. For each parallel goal, if it was not suspended when performing speculative backtracking, its last answer is already on the execution envi-

ronment ready to be combined. Otherwise, its first answer is reinstalled on the heap before continuing with forward execution.

When there is more than one possible answer combination (because some parallel goals already found more than one answer), a *ghost* choice point is created. This choicepoint has an “artificial” alternative which points to code which takes care of retrieving saved answers and installing the bindings. On backtracking, this code will produce the combinations of answers triggered by the newly found answer (i.e., combinations already produced are not repeated). Note that this new answer may have been produced by any goal in the conjunction, but we proceed by combining from right to left. The invariant here is that before producing a new answer, all previous answer combinations have been produced, so we only need to fix the bindings for the goal which produced the new answer (say g) and successively installing the bindings for the saved answers produced by the rest of the goals.

Therefore, we start by installing one by one the answers previously produced by the rightmost goal. When all solutions are exhausted, we move on to the next goal to the left, install its next answer and then reinstall again one by one the answers of the rightmost goal. When all the combinations of answers for these two goals are exhausted, we move on to the third rightmost one, and so on —but we skip goal g, because we only need to combine its last answer since the previous ones were already combined.

An additional optimization is to update the heap top pointer of the *ghost* choice point to point to the current heap top after copying terms from the memoization area to the heap, in order to protect these terms from backtracking for a possible future answer combination. Consequently, when the second answer of the second rightmost parallel goal is combined with all the answers of the rightmost goal, the bindings of the answers of the rightmost goal do not need to be copied on the heap again and then we only need to untrail bindings from the last combined answer and redo bindings of the answer being combined. Finally, once the *ghost* choice point is eliminated, all these terms that were copied on the heap are released.

One particular race situation needs to be considered. When a parallel goal generates a new solution, other parallel goals may also find new answers before being suspended, and thus some answers may be lost in the answer combination. In order to address this, our implementation maintains a pointer to the last combined answer of each parallel goal in the parcall frame. Therefore, if, e.g., two parallel goals, `a/1` and `b/1`, have computed three answers each, but only two of them have been combined, the third answer of `a/1` would be combined with the first two answers of `b/1`, updating afterward its last combined answer pointer to its third answer. Once this is done, the fact that `b/1` has uncombined answers is detected before performing backtracking, and the third answer of `b/1` is combined with all the computed answers of `a/1` and, then, the last combined answer of `b(Y)` is updated to point to its last answer. Finally, when no goal is left with uncombined answers, the answer combination operation fails.

5 Trapped Goals and Backtracking Order

The classical, right-to-left backtracking order for IAP is known to bring a number of challenges, among them the possibility of *trapped goals*: a goal on which backtracking has to be performed becomes *trapped* by another goal stacked on top of it. Normal backtracking is therefore impossible. Consider the following example:

```
m(X,Y,Z) :- b(X,Y) & a(Z).
b(X,Y) :- a(X) & a(Y).
a(1).  a(2).
```

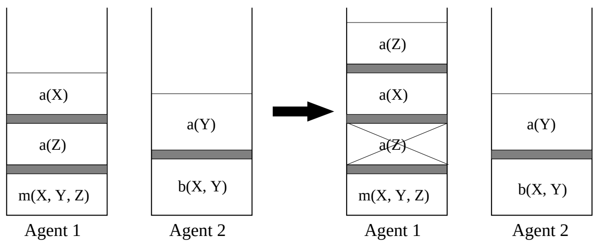


Fig. 3. Execution of m/3.

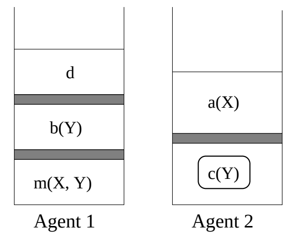


Fig. 4. Execution of m/2.

Figure 3 shows a possible state of the execution of predicate m/3 by two agents. When the first agent starts computing m/3, b(X, Y) and a(Z) are scheduled to be executed in parallel. Assume that a(Z) is executed locally by the first agent and b(X,Y) is executed by the second agent. Then, the second agent schedules a(X) and a(Y) to be executed in parallel, which results in a(Y) being locally executed by the second agent and a(X) executed by the first agent after computing an answer for a(Z). In order to obtain another answer for m/3, right-to-left backtracking requires computing additional answers for a(Z), a(Y), and a(X), in that order. However, a(Z) cannot be directly backtracked over since a(X) is stacked on top of it: a(Z) is a *trapped goal*.

Several solutions have been proposed for this problem. One of the original proposals uses *continuation markers* (Hermenegildo 1986; Shen and Hermenegildo 1996) to *skip* over stacked goals. This is, however, difficult to implement properly and needs to take care of a large number of cases. It can also leave unused sections of memory (*garbage slots*) which are either only reclaimed when finally backtracking over the parallel goals, or require quite delicate memory management. A different solution (Casas et al. 2008) is to move the execution of the trapped goal to the top of the stack. This simplifies the implementation somewhat, but it also leaves garbage slots in the stacks.

5.1 Out-of-Order Backtracking

Our approach does not follow the sequential backtracking order, to reduce the likelihood of the appearance of trapped goals and garbage slots. The key idea is to allow backtracking (and therefore the order of solutions) to dynamically adapt to the configuration of the stacks.

As mentioned before, the obvious drawback of this approach is that it may alter solution order with respect to sequential execution, and in an unpredictable way. However, we argue that in many cases this may not be a high price to pay, specially if the programmer is aware of it and can have a choice. Programs where solution order matters, typically because of efficiency, are likely to have dependencies between goals which would anyway make them not amenable for IAP. For independent goals we argue that allowing out-of-order backtracking represents in some way a return to a simpler, more declarative semantics that has the advantage of allowing higher efficiency in the implementation of parallelism.

The alternative we propose herein consists of always backtracking over the goal that is on top of the stack, without taking into account the original goal execution order. For example, in the case of backward execution over predicate m/3 in Figure 3, both agents may be able to backtrack over a(X) and a(Y), without having to move the execution of a(Z).

5.2 First Answer Priority and Trapped goals

Out-of-order backtracking, combined with answer memoing not to lose answer combinations, can avoid trapped goals if no priority is given to any of the parallel goals, because there will always be a backtrackable goal on the stack top to continue the execution of the program. However, as mentioned before, we do impose a lightweight notion of priority to first answers to preserve no-slowdown: backward execution of parallel goals that have not found any answer has more priority than backward execution of parallel goals which have already found an answer. Note that even using this very lax notion of priority, the possibility of trapped goals returns, as illustrated in the following example:

```
m(X,Y) :- a(X) & b(Y).
b(Y) :- c(Y) & d,   e(Y).
a(1).  a(2).  c(1).  c(2).  d.  e(2)
```

Figure 4 shows a possible state of the execution of predicate m/2 by two agents. The first agent starts with the execution of predicate m/2 and publishes a/1 and b/1 to be executed in parallel. The first agent starts with the execution of b/1 and marks both c/1 and d/0 for parallel execution. The second agent then executes c/1 while the first agent is executing d/0, and when the execution of c/1 finishes then it computes an answer for a/1. Once the execution of goals c/1 and d/0 has finished, e/1 is executed. However, this execution will fail because c/1 already gave a different binding to variable Y. If the first answer is given priority, c/1 should be backtracked before a/1, but c/1 is trapped by the execution of a/1. While this example shows that it is possible to have trapped goals with out-of-order backtracking, we experimentally found that the percentage of trapped goals vs. remotely executed goals varies between 20% and 60% under right-to-left backtracking and it is always 0% under out-of-order backtracking, thus allowing for a simpler solution for the problem without degrading the performance of parallel execution.

Our approach is to perform stack reordering to create a new execution state which is consistent, i.e., which could have been generated by a sequential SLD execution. Consequently, the parallel scheduler is greatly simplified since it does not have to manage trapped goals. We cannot present the algorithm due to space limitations, but a high-level view follows:

1. Copy the choice point and trail section corresponding to the trapped goal to the top of the stacks (their original allocations become garbage).
2. Move down the choice point and trail section to remove the generated garbage slots.
3. Update the trail pointers of relocated choice points to the reordered trail section.
4. Keep heap and local stack in the same location. Global and frame stack top pointers of the trapped goal choice points are updated to point to the actual top of global and frame stack. Consequently, the execution memory of the goals that were moved down the stack is protected from backtracking.

6 The Scheduler for the Parallel Backtracking IAP Engine

Once we allow backward execution over any parallel goal on the top of the stacks, we can perform backtracking over all of them in parallel. Consequently, each time we perform backtracking over a parallel conjunction, each of the parallel goals of the parallel conjunction can start speculative backward execution.

As we mentioned earlier, the management of goals (when a goal is available and can start, when it has to backtrack, when messages have to be broadcast, etc.) is encoded in Prolog

```
parcall_back(LGoals, NGoals) :-
    fork(PF,NGoals,LGoals,[Handler|LHandler]),
    (
        goal_not_executed(Handler) ->
        call_local_goal(Handler,Goal)
    ;
        true
    ),
    look_for_available_goal(LHandler),
    join(PF).

look_for_available_goal([]) :-   !,  true.
look_for_available_goal([Handler|LHandler])  :-
    (
        goal_available(Handler) ->
        call_local_goal(Handler,Goal)
    ;
        true
    ),
    look_for_available_goal(LHandler).
```

```
agent :- work, agent.
agent :- agent.

work :-
    find_parallel_goal(Handler)  ->
    (
        goal_not_executed(Handler) ->
            save_init_execution(Handler),
            call_parallel_goal(Handler)
        ;
            move_execution_top(Handler),
            fail
    )
  ;
    suspend,
    work.
```

Fig. 5. Parallel backtracking Prolog code.

code which interacts with the internals of the emulator. Figure 5 shows a simplified version of such a scheduler, which is executed when agents (a) look for new work to do and (b) have to execute a parallel conjunction. Note that locks are not shown in the algorithm.

6.1 Looking for Work

Agents initially execute the agent/0 predicate, which calls work/0 in an endless loop to search for a parallel goal to execute, via the find_parallel_goal/1 primitive, which defines the strategy of the scheduler. Available goals can be in four states: non-executed parallel goals necessary for forward execution, backtrackable parallel goals necessary for forward execution, non-executed parallel goals not necessary for forward execution (because they were generated by goals performing speculative work), and backtrackable parallel goals not necessary for forward execution. Different scheduling policies are possible in order to impose preferences among these types of goals (to, e.g., decide which non-necessary goal can be picked) but studying them is outside the scope of this paper.

Once the agent finds a parallel goal to execute, it is prepared to start execution in a clean environment. For example, if the goal has to be backtracked over and it is trapped, a primitive operation move_execution_top/1 moves the execution segment of the goal to the top of the stacks to ensure that the choice point to be backtracked over is always on the top of the stack (using the algorithm of Section 5). Also, the memoization of the last answer found is performed at this time, if the execution of the parallel goal was not suspended.

If find_parallel_goal/1 fails (i.e., no handler is returned), the agent suspends until some other agent publishes more work. call_parallel_goal/1 saves some registers before starting the execution of the parallel goal, such as the current trail and heap top, changes the state of the handler once the execution has been completed, failed, or suspended, and saves some registers after the execution of the parallel goal in order to manage trapped goals and to release the execution of the publishing agent.

6.2 Executing Parallel Conjunctions

The parallel conjunction operator &/2 is preprocessed and converted into parcall_back/2, which is the entry point of the scheduler, and which receives the list of goals to execute in parallel (LGoals) and the number of goals in the list. parcall_back/2 invokes first fork/4, written in C, which creates a *handler* for each parallel goal in the scope of the parcall frame containing information related to that goal, makes goals available for other agents to pick up, resumes suspended agents which can then steal some of the new available goals, and inserts a new choice point in order to release all the data structures on failure.

If the first parallel goal has not been executed yet, it is scheduled for local execution by call_local_goal/2, which performs housekeeping similar to that of call_parallel_goal/1. It can be already executed because this parallel goal, which is always executed locally, can fail on backtracking, but the rest of the parallel goals could still be performing backtracking to compute more answers. In this case, the choice point of fork/4 will succeed on backtracking to continue forward execution and to wait for the completion of the remotely executed parallel goals to produce more answer combinations.

Then, look_for_available_goal/1 executes locally parallel goals which have not already been taken by another agent. Finally, join/1 waits for the completion of the execution of the parallel goals, their failure, or their suspension before combining all the answers. After all answers have been combined, the goals of the parallel conjunction are activated to perform speculative backward execution.

7 Suspension of Speculative Goals

Stopping goals which are eagerly generating new solutions may be necessary for both correctness and performance reasons. The agent that determines that suspension is necessary sends a suspension event to the rest of the agents that stole any of the sibling parallel goals (accessible via the parcall frame). These events are checked in the WAM loop each time a new predicate is called, using existing event-checking machinery shared with attributed-variable handling (and therefore no additional overhead is added). When the execution has to suspend, the argument registers are saved on the heap, and a new choice point is inserted onto the stack to protect the current execution state. This choice point contains only one argument pointing to the saved registers in order to reinstall them on resumption. The alternative to be executed on failure points to a special WAM instruction which reinstalls the registers and jumps to the WAM code where the suspension was performed, after releasing the heap section used to store the argument registers. Therefore, the result of failing over this choice point is to resume the suspended execution at the point where it was suspended.

After this choice point is inserted, goal execution needs to jump back to the Prolog scheduler for parallel execution. In order to jump to the appropriate point in the Prolog scheduler (after call_parallel_goal/1 or call_local_goal/2), the WAM frame pointer is saved in the handler of the parallel goal before calling call_parallel_goal/1 or call_local_goal/2. After suspension takes place, it is reinstalled as the current frame pointer, the WAM's *next instruction* pointer is updated to be the one pointed to by this frame, and this WAM instruction is dispatched. The result is that the scheduler continues its execution as if the parallel goal had succeeded.

Parallel goals to be suspended may in turn have other nested parallel calls. Suspension events are recursively sent by agents following the chain of dependencies saved in the parcall frames, similarly to the *fail* messages in &-Prolog (Hermenegildo and Greene 1991).

8 A Note on Deterministic Parallel Goals

The machinery we have presented can be greatly simplified when running deterministic goals in parallel: answer memoization and answer combination are not needed, and the scheduler (Section 6) can be simplified. Knowing ahead of execution which goals are deterministic can be used to statically select the best execution strategy. However, some optimizations can be performed dynamically without compiler support (e.g., if it is not available or imprecise). For example, the move_execution_top/1 operation may decide not to memoize the previous answer if there are no choice points associated to the execution of the parallel goal, because that means that at most one answer can be generated. By applying these dynamic optimizations, we have detected improvements of up to a factor of two in the speedups of the execution of some deterministic benchmarks.

9 Comparing Performance of IAP Models

We present here a comparison between a previous high-level implementation of IAP (Casas et al. 2008) (which we abbreviate as seqback) with our proposed implementation (parback). Both implementations are similar in nature and have similar overheads (inherent to a high-level implementation), with the obvious main difference being the support for parallel backtracking and answer memoization in parback. Both are implemented by modifying the standard Ciao (Bueno et al. 2009; Hermenegildo et al. 2011) distribution. We will also comment on the relation with the very efficient IAP implementation in (Hermenegildo and Greene 1991) (abbreviated as &-Prolog) for deterministic benchmarks in order to evaluate the overhead incurred by having part of the system expressed in Prolog.

We measured the performance results of both parback and seqback on deterministic benchmarks, to determine the possible overhead caused by adding the machinery to perform parallel backtracking and answer memoization, and also of course on non-deterministic benchmarks. The deterministic benchmarks used are the well-known Fibonacci series (*fibo*), matrix multiplication (*mmat*) and QuickSort (*qsort*). *fibo* generates the 22^{nd} Fibonacci number switching to a sequential implementation from the 12^{th} number downwards, *mmat* uses 50x50 matrices and *qsort* is the version which uses append/3 sorting a list of 10000 numbers. The GC suffix means task granularity control (López-García et al. 1996) is used for lists of size 300 and smaller.

The selected nondeterministic benchmarks are *checkfiles*, *illumination*, and *qsort_nd*. *checkfiles* receives a list of files, each of which contains a list of file names which may exist or not. These lists are checked in parallel to find nonexistent files which appear listed in all the initial files; these are enumerated on backtracking. *illumination* receives an $N \times N$ board informing of possible places for lights in a room. It tries to place a light in each of the columns, but lights in consecutive columns have to be separated by a minimum distance. The eligible positions in each column are searched in parallel and position checking is implemented with a pause of one second to represent task lengths. *qsort_nd* is a QuickSort algorithm where list elements have only a partial order. *checkfiles* and *illumination* are synthetic benchmarks which create 8 parallel goals and which exploit memoization heavily. *qsort_nd* is a more realistic benchmark which creates over one thousand parallel goals. All the benchmarks were parallelized using CiaoPP (Hermenegildo et al. 2005) and the annotation algorithms described in (Muthukumar et al. 1999; Cabeza 2004; Casas et al. 2007).

Table 1 shows the speedups obtained. Performance results for seqback and parback were obtained by averaging ten different runs for each of the benchmarks in a Sun Ultra-

Benchmark	Approach	Number of threads							
		1	2	3	4	5	6	7	8
Fibo	&-Prolog	0.98	1.93	-	3.70	-	5.65	-	7.34
	seqback	0.95	1.89	2.80	3.70	4.61	5.36	6.23	6.96
	parback	0.95	1.88	2.78	3.69	4.60	5.33	6.21	6.94
	parback$_{det}$	0.96	1.91	2.83	3.74	4.65	5.41	6.28	7.04
QSort	&-Prolog	1.00	1.92	-	3.03	-	3.89	-	4.65
	seqback	0.50	0.98	1.38	1.74	2.05	2.27	2.57	2.67
	parback	0.49	0.97	1.37	1.74	2.05	2.27	2.58	2.69
	parback$_{det}$	0.56	1.10	1.54	1.96	2.31	2.57	2.90	3.02
	seqbackGC	0.97	1.77	2.42	3.02	3.37	3.77	3.98	4.15
	parbackGC	0.97	1.76	2.41	3.00	3.34	3.74	3.94	4.12
	parbackGC$_{det}$	0.97	1.78	2.44	3.04	3.41	3.79	3.99	4.21
MMat	&-Prolog	1.00	1.99	-	3.98	-	5.96	-	7.93
	seqback	0.78	1.55	2.28	2.99	3.67	4.29	4.91	5.55
	parback	0.76	1.52	2.25	2.95	3.60	4.22	4.83	5.45
	parback$_{det}$	0.80	1.60	2.38	3.01	3.79	4.55	5.19	5.87
CheckFiles	seqback$_{first}$	0.99	1.09	1.11	1.12	1.12	1.12	1.13	1.13
	seqback$_{all}$	0.99	1.05	1.07	1.07	1.07	1.08	1.08	1.08
	parback$_{first}$	3917	8612	10604	17111	17101	17116	17134	44222
	pb_rel$_{first}$	1.00	2.20	2.71	4.37	4.37	4.37	4.37	11.29
	parback$_{all}$	12915	23409	30545	45818	46912	46955	46932	89571
	pb_rel$_{all}$	1.00	1.81	2.37	3.55	3.63	3.64	3.63	6.94
Illumination	seqback$_{first}$	1.00	1.37	1.55	1.56	1.56	1.61	1.67	1.67
	seqback$_{all}$	1.00	1.16	1.21	1.24	1.24	1.25	1.25	1.27
	parback$_{first}$	1120	1725	2223	3380	3410	4028	4120	6910
	pb_rel$_{first}$	1.00	1.54	1.98	3.02	3.04	3.60	3.68	6.17
	parback$_{all}$	8760	16420	20987	31818	31912	31888	31934	65314
	pb_rel$_{all}$	1.00	1.87	2.40	3.63	3.64	3.64	3.65	7.46
QSortND	seqback$_{first}$	0.94	1.72	2.36	2.92	3.25	3.59	3.78	3.92
	seqback$_{all}$	0.91	0.96	0.98	0.99	0.99	1.00	1.00	1.00
	parback$_{first}$	0.94	1.72	2.35	2.91	3.24	3.57	3.76	3.91
	parback$_{all}$	4.29	6.27	8.30	9.90	10.5	10.9	11.1	11.3
	pb_rel$_{all}$	1.00	1.46	1.93	2.31	2.45	2.54	2.59	2.64

Table 1. Comparison of speedups for several benchmarks and implementations.

Sparc T2000 (a *Niagara*) with 8 4-thread cores. The speedups shown in this table are calculated with respect to the sequential execution of the original, unparallelized benchmark. Therefore, the column tagged 1 corresponds to the slowdown coming from executing a parallel program on a single processor. For &-Prolog we used the results in (Hermenegildo and Greene 1991). To complete the comparison, we note that one of the most efficient Prolog systems, YAP Prolog (**?**), very optimized for SPARC, is on these benchmarks between 2.3 and 2.7 faster than the execution of the parallel versions of the programs on the parallel version of Ciao using only one agent, but the parallel execution still outperforms YAP. Of course, YAP could in addition take advantage of parallel execution.

For deterministic benchmarks, parback$_{det}$ refers to the implementation presented in this paper with improvements based on determinacy information obtained from static analysis (López-García et al. 2005). For nondeterministic benchmarks we show a comparison of the performance results obtained both to generate the first solution (seqback$_{first}$ and parback$_{first}$) and all the solutions (seqback$_{all}$ and parback$_{all}$). Additionally, we also show speedups relative to the execution in parallel with memoing in one agent (which should be similar to that which could be obtained by executing sequentially with memoing) in rows pb_rel$_{first}$ and pb_rel$_{all}$.

The speedups obtained in both high-level implementations are very similar for the case of deterministic benchmarks. Therefore, the machinery necessary to perform parallel backtracking does not seem to degrade the performance of deterministic programs.

Static optimizations bring improved performance, but in this case they seem to be quite residual, partly thanks to the granularity control. When comparing with &-Prolog we of course suffer from the overhead of executing partly at the Prolog level (especially in mmat and qsort without granularity control), but even in this case we think that our current implementation is competitive enough. It is important that to note that the &-Prolog speedups were measured in another architecture (Sequent Symmetry), so the comparison can only be indicative. However, the Sequents were very efficient and orthogonal multiprocessors, probably better than the Niagara in terms of obtaining speedups (even if obviously not in raw speed) since the bus was comparatively faster in relation with processor speed. This can only make &-Prolog (and similar systems) have smaller speedups if run in parallel hardware. Therefore, their speedup could only get closer to ours in current architectures.

parback and seqback behavior is quite similar in the case of *qsort_nd* when only the first answer is computed because there is not backtracking here.

In the case of *checkfiles* and *illumination*, backtracking is needed even to generate the first answer, and memoing plays a more important role. The implementation using parallel backtracking is therefore much faster even in a single processor since recomputation is avoided. If we compute the speedup relative to the parallel execution on one processor (rows pb_rel$_{first}$ and pb_rel$_{all}$) the speedups obtained by parback follow the increment in the number of processors more closely —with some superlinear speedup which is normal when search does not follow, as in our case, the same order as sequential execution— which can be traced to the increased amount of parallel backtracking. In contrast, the speedups of seqback do not increase so much since it performs essentially sequential backtracking.

When all the answers are required, the differences are still clearer because there is much backward execution. This behavior also appears, to a lesser extent, in *qsort_nd*. More in detail, the parback speedups are not that good when looking for all the answers of *qsort_nd* because the time for storing and combining answers is not negligible here.

Note that the parback speedups of *checkfiles* and *illumination* stabilize between 4 and 7 processors. This is so because they generate exactly 8 parallel goals, and there is one dangling goal to be finished. In the case of *checkfiles* we get superlinear speedup because there are 8 lists of files to check. With 8 processors the first answer can be obtained without traversing (on backtracking) any of these lists. This is not the case with 7 processors and so there is no superlinear behavior until we hit the 8 processor mark. Additionally, since backtracking is done in parallel, the way the search tree is explored (and therefore how fast the first solution is found) can change between executions.

10 Conclusions

We have developed a parallel backtracking approach for independent and-parallelism which uses out-of-order backtracking and relies on answer memoization to reuse and combine answers. We have shown that the approach can bring interesting simplifications when compared to previous approaches to the complex implementation of the backtracking mechanism typical in these systems. We have also provided experimental results that show significant improvements in the execution of non-deterministic parallel calls due to the avoidance of having to recompute answers and due to the fact that parallel goals can execute backward in parallel, which was a limitation in previous similar implementations. This parallel system may be used in applications with a constraint-and-generate structure in which checking the restrictions after the search is finished does not add significant computation, and a simple code transformation allows a sequential program to be executed in parallel.

References

AIT-KACI, H. 1991. *Warren's Abstract Machine, A Tutorial Reconstruction*. MIT Press.

ALI, K. A. M. AND KARLSSON, R. 1990. The Muse Or-Parallel Prolog Model and its Performance. In *1990 North American Conference on Logic Programming*. MIT Press, 757–776.

BUENO, F., CABEZA, D., CARRO, M., HERMENEGILDO, M., LÓPEZ-GARCÍA, P., AND PUEBLA-(EDS.), G. 2009. The Ciao System. Ref. Manual (v1.13). Tech. rep., School of Computer Science, T.U. of Madrid (UPM). Available at `http://www.ciaohome.org`.

CABEZA, D. 2004. An Extensible, Global Analysis Friendly Logic Programming System. Ph.D. thesis, Universidad Politécnica de Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain.

CASAS, A., CARRO, M., AND HERMENEGILDO, M. 2007. Annotation Algorithms for Unrestricted Independent And-Parallelism in Logic Programs. In *17th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'07)*. Number 4915 in LNCS. Springer-Verlag, The Technical University of Denmark, 138–153.

CASAS, A., CARRO, M., AND HERMENEGILDO, M. 2008. A High-Level Implementation of Non-Deterministic, Unrestricted, Independent And-Parallelism. In *24th International Conference on Logic Programming (ICLP'08)*, M. García de la Banda and E. Pontelli, Eds. LNCS, vol. 5366. Springer-Verlag, 651–666.

CHEN, W. AND WARREN, D. S. 1996. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM 43,* 1 (January), 20–74.

CONERY, J. S. 1987. *Parallel Execution of Logic Programs*. Kluwer Academic Publishers.

COSTA, V. S., DAMAS, L., REIS, R., AND AZEVEDO, R. 2002. *YAP User's Manual*. `http://www.dcc.fc.up.pt/~vsc/Yap`.

GUPTA, G., PONTELLI, E., ALI, K., CARLSSON, M., AND HERMENEGILDO, M. 2001. Parallel Execution of Prolog Programs: a Survey. *ACM Transactions on Programming Languages and Systems 23,* 4 (July), 472–602.

HERMENEGILDO, M. 1986. An abstract machine based execution model for computer architecture design and efficient implementation of logic programs in parallel. Ph.D. thesis, U. of Texas at Austin.

HERMENEGILDO, M. AND GREENE, K. 1991. The &-Prolog System: Exploiting Independent And-Parallelism. *New Generation Computing 9,* 3,4, 233–257.

HERMENEGILDO, M., PUEBLA, G., BUENO, F., AND LÓPEZ-GARCÍA, P. 2005. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming 58,* 1–2.

HERMENEGILDO, M. AND ROSSI, F. 1995. Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions. *Journal of Logic Programming 22,* 1, 1–45.

HERMENEGILDO, M. V., BUENO, F., CARRO, M., LÓPEZ, P., MERA, E., MORALES, J., AND PUEBLA, G. 2011. An Overview of Ciao and its Design Philosophy. *Theory and Practice of Logic Programming*. http://arxiv.org/abs/1102.5497.

JANSON, S. 1994. Akl. a multiparadigm programming language. Ph.D. thesis, Uppsala University.

KARP, A. AND BABB, R. 1988. A Comparison of 12 Parallel Fortran Dialects. *IEEE Software*.

LOPES, R., SANTOS COSTA V., AND SILVA, F. M. A. 2011. A Design and Implementation of the Extended Andorra Model. *Theory and Practice of Logic Programming*.

LÓPEZ-GARCÍA, P., BUENO, F., AND HERMENEGILDO, M. 2005. Determinacy Analysis for Logic Programs Using Mode and Type Information. In *Proceedings of the 14th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'04)*. Number 3573 in LNCS. Springer-Verlag, 19–35.

LÓPEZ-GARCÍA, P., HERMENEGILDO, M., AND DEBRAY, S. K. 1996. A Methodology for Granularity Based Control of Parallelism in Logic Programs. *Journal of Symbolic Computation, Special Issue on Parallel Symbolic Computation 21,* 4–6, 715–734.

LUSK, E., BUTLER, R., DISZ, T., OLSON, R., STEVENS, R., WARREN, D. H. D., CALDERWOOD, A., SZEREDI, P., BRAND, P., CARLSSON, M., CIEPIELEWSKI, A., HAUSMAN, B., AND HARIDI, S. 1988. The Aurora Or-parallel Prolog System. *New Generation Computing 7,* 2/3, 243–271.

MOURA, P., CROCKER, P., AND NUNES, P. 2008. High-level multi-threading programming in logtalk. In *10th International Symposium on Practical Aspects of Declarative Languages (PADL'08)*, D. Warren and P. Hudak, Eds. LNCS, vol. 4902. Springer-Verlag, 265–281.

MUTHUKUMAR, K., BUENO, F., DE LA BANDA, M. G., AND HERMENEGILDO, M. 1999. Automatic Compile-time Parallelization of Logic Programs for Restricted, Goal-level, Independent And-parallelism. *Journal of Logic Programming 38,* 2 (February), 165–218.

PONTELLI, E., GUPTA, G., AND HERMENEGILDO, M. 1995. &ACE: A High-Performance Parallel Prolog System. In *International Parallel Processing Symposium*. IEEE Computer Society Technical Committee on Parallel Processing, IEEE Computer Society, 564–572.

RAMAKRISHNAN, I., RAO, P., SAGONAS, K., SWIFT, T., AND WARREN, D. 1995. Efficient tabling mechanisms for logic programs. In *ICLP*. 697–711.

SANTOS-COSTA, V. M. 1993. Compile-time analysis for the parallel execution of logic programs in andorra-i. Ph.D. thesis, University of Bristol.

SANTOS COSTA, V., WARREN, D., AND YANG, R. 1991. The Andorra-I Engine: A Parallel Implementation of the Basic Andorra Model. In *ICLP*. 825–839.

SHEN, K. 1996. Overview of DASWAM: Exploitation of Dependent And-parallelism. *Journal of Logic Programming 29,* 1–3 (November), 245–293.

SHEN, K. AND HERMENEGILDO, M. 1996. Flexible Scheduling for Non-Deterministic, And-parallel Execution of Logic Programs. In *Proceedings of EuroPar'96*. Number 1124 in LNCS. Springer-Verlag, 635–640.

TAMAKI, H. AND SATO, M. 1986. OLD resolution with tabulation. In *Int'l. Conf. on Logic Programming*. LNCS, Springer-Verlag, 84–98.

WARREN, D. 1983. An Abstract Prolog Instruction Set. Technical Report 309, Artificial Intelligence Center, SRI International, 333 Ravenswood Ave, Menlo Park CA 94025.

WARREN, D. S. 1992. Memoing for logic programs. *Communications of the ACM 35,* 3, 93–111.